PeerCQ: A Scalable and Self-Configurable Peer-to-Peer Information Monitoring System

Buĵra Gedik
Georgia Institute of Technology
College of Computing
Atlanta, GA 30332, USA
bgedik@cc.gatech.edu

Ling Liu
Georgia Institute of Technology
College of Computing
Atlanta, GA 30332, USA
lingliu@cc.gatech.edu

Abstract

PeerCQ is a peer-to-peer Continual Query system for information monitoring on the Internet. It uses Continual Queries (CQs) as its primitives to express information-monitoring requests. A primary objective of the PeerCQ system is to build a decentralized Internet scale distributed information-monitoring system, which is highly scalable, self-configurable and supports efficient and robust way of processing CQs. In this paper we describe the basic architecture of the PeerCQ system and focus on the mechanisms used for service partitioning and service lookup. There are two unique characteristics of PeerCQ. First, it introduces a donation based peer-aware mechanism for handling the peer heterogeneity. Second, it integrates CQ-aware and peer-aware information into its service partitioning scheme, while maintaining decentralization and self-configurability. We report a set of initial experiments demonstrating the sensitiveness of our approach to peer heterogeneity and the effectiveness of our service partitioning algorithm with respect to load balancing and system utilization.

1 Introduction

Peer-to-peer (P2P) systems become increasingly popular with the emergence of successful applications like Gnutella [8], Napster [10] and Freenet [1]. The motivating idea behind the P2P approach in these applications is two folds: to reduce the role of centralized servers which are the main points of failure, and to allow users around the world to communicate and share their resources and services, utilizing the otherwise lost processor cycles.

P2P systems have their own challenges. Recognizing the diversity of the peers (differences in their resources, load characteristics, trust of service, and service times) and utilizing such information to distribute the work among the peers is a big challenge. Building a P2P architecture that handles naming and resource discovery while having good performance and supporting the integration of security and anonymity is another challenge.

Most of the P2P applications built today deal with storage sharing to exchange data (like [1,8,10]) or distributed computing to solve large computation problems (like [9,14]). An interesting application we are building is PeerCQ, a peer-to-peer information monitoring system.

PeerCQ is a decentralized Internet scale P2P continual query system for monitoring information change on the web. It uses continual queries as its primitives to express information monitoring requests. Continual Queries (CQs) [3] are standing queries that monitor information updates and return results whenever the updates reach certain specified thresholds. There are three main components of a CQ: query, trigger, and stop condition. Whenever the trigger condition becomes true, the query part is executed and the part of the query result that is different from the result of the previous execution is returned. The stop condition specifies the termination of a CQ. It is important to note that continual queries are defined over past, present, and future data. Two important properties of CQ domain need special consideration when designing a CQ system. First, CQs are long running entities. It is not acceptable to break a CQ execution and resume it at some arbitrary time or run it over from the beginning. Once started a CQ has to run until its stop condition is reached. Second and more importantly, continual queries should be grouped based on the similarity of their triggers to avoid or reduce the amount of repeated computation and ensure an efficient and scalable processing of CQs. Both characteristics pose additional challenges in a P2P information monitoring system, such as PeerCQ.

PeerCQ is an interesting application for two reasons: First, it has an alternative approach to client/server based continual query systems [3,4]. It uses a totally decentralized P2P architecture. It distributes CQs that are long running entities, to the nodes of the system in order to optimize the execution efficiency. PeerCQ is more complex than basic P2P file sharing applications. It can be seen as a P2P distributed computing application. But it is different than the conventional P2P distributed computing applications in the sense that it does not try to solve one large problem by partitioning it into parts and assigning each part to a node. In contrast PeerCQ system tries to execute lots of long running jobs by ensuring each job is assigned to a node at any given time. The complexity of the CQ domain differentiates PeerCQ from other P2P distributed computing applications.

Second, PeerCQ poses several technical challenges in providing information monitoring services using a P2P computing paradigm. The first challenge is the need of a *smart service partitioning* mechanism. The main issue regarding service partitioning is the load balance among peers of the system. Several factors can affect the load balancing decision, including the willingness of peers to participate, the computing capacity and the desired resource contribution of the peers, and the characteristics of the continual queries. The effectiveness of such load balancing decision has an impact on overall scalability of the system. Other technical challenges include failure handling and trust management. In PeerCQ, failure handling mechanisms are needed to detect failures and ensure correct and smooth CQ execution, and to cope with unknown peers or unpredictability of the peers. Trust is an important measure of the stableness of the peers. Having methods to integrate the trust in service partitioning is important, since peers that enter and leave the system very frequently are not useful in collaborative CQ processing. Another issue that needs consideration is the protection mechanisms needed against malicious peers.

In this paper, we focus on the problems related to service partitioning, and our technical solution to address these problems. A unique feature of the PeerCQ service partitioning mechanism is its ability to integrate both node/peer information and user/CQ information into the load balancing scheme, a challenge in all large-scale, heterogeneous, and totally decentralized systems. PeerCQ's load balancing scheme takes into account both the peer heterogeneity and the user characteristics in order to provide more robust and scalable support for information monitoring over a P2P network. In comparison, most of the current P2P protocols [1,2,5,6,8] to date make the assumption that all nodes tend to participate and contribute equally to the system, and assign responsibilities to peers based on this assumption. Work done for analyzing characteristics of Napster and Gnutella in [15] shows that peers participating in these systems are heterogeneous with respect to many characteristics, such as connection speeds, shared disk space, and peers' willingness to participate. These evidences show that P2P applications should respect the peer heterogeneity and user characteristics in order to be more robust [15].

In summary, this paper has two main contributions. First, we introduce a distinct approach to CQ systems, which enables processing of large number of CQs by harnessing the power at the edge of the Internet, without a need for centralized servers. Second, we introduce a smart service partitioning scheme. It improves current P2P protocols by integrating both user and peer information into service partitioning decisions. PeerCQ's service partitioning scheme aims at balancing the load in the presence of peer heterogeneity, and enhancing system scalability in the presence of large number of peers and large number of triggers firing over many data sources.

2 Architecture Overview

PeerCQ is a large-scale peer-to-peer system for information monitoring on the Internet. Peers are machines on the Internet that execute PeerCQ servant application. The term servant expresses that the peers of PeerCQ are acting both as clients and servers.

There are three main mechanisms that make up the PeerCQ system. The first mechanism is the overlay network membership. A peer that wants to join to the system should be integrated into the PeerCQ overlay network that is formed by the peers already participating in the system and a peer that wants to depart (disconnection is treated same as departure) from the system should be removed from the PeerCQ overlay network. The second mechanism is the smart service partitioning. In PeerCQ every peer participates in the process of evaluating CQs. A peer joining the PeerCQ network is assigned some CQs to process, and it can post a new CQ of its own interest. When a new CQ is posted, the peer that will process it should be chosen in a way that balances the load on peers, in the sense that no peers will be overloaded. Moreover, upon a peer's entrance to the system, a set of CQs that needs to be assigned to this peer should be determined by again taking into account the goal of load balancing. Similarly, when a peer departs from the system, the set of CQs responsible by it should be reassigned to the rest of peers, while maintaining the load balancing objective. The third mechanism is the CQ processing. CQs should be executed at their assigned peers and be cleanly migrated to other peers in the presence of failure or peer entrance and departure.

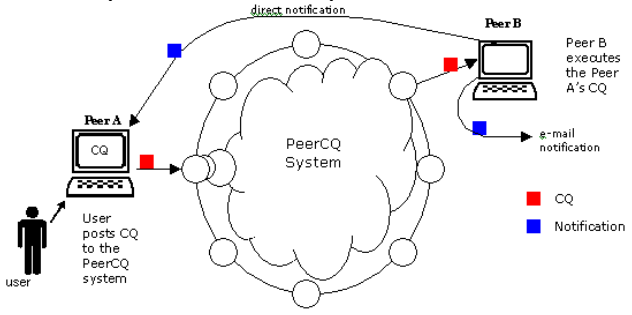

Figure 1

The general scenario in PeerCQ from the user's point of view is depicted in Figure 1. A user composes a CQ and posts it to the PeerCQ system via a peer, say peer A. Based on the PeerCQ's service partition scheme (see Section 5), Peer A is not responsible for this CQ. Thus it triggers the PeerCQ's service lookup function. The PeerCQ system determines which peer will be responsible for processing this CQ. Assume that the peer B was chosen to execute this CQ. After the CQ is assigned to the peer B, it starts execution. During the execution of this CQ, when the peer B detects an interested information update and runs the query, it notifies the owner of this CQ with the newly updated information. This notification could be realized by e-mail or by directly sending it to the

peer A if it is online at the time of notification. Note that, even if a peer is not participating in the system at a given time, its previously posted CQs are in execution at other places.

3 P2P Protocol Design Alternatives

A common problem in P2P systems is to find peers that can satisfy a request of another peer. In a file sharing system this is the problem of locating peers that has the specified file. In a messaging system, this is the problem of locating the peer that has the specified profile or application specific name. In a distributed computing system, this is the problem of locating the peer that is going to execute a specified piece of job. And in the PeerCQ system, this is the problem of locating the peer that is going to execute a CQ posted by another peer. Generally, this can be viewed as a lookup problem. Different P2P protocols have different solutions to the lookup problem.

A number of P2P lookup approaches have been proposed. We classify them into three categories, and discuss each of the categories in terms of their respective P2P protocol and their applicability to the PeerCQ system.

3.1 Centralized look-up

In the centralized lookup approach each peer connects to the centralized server and registers itself. Queries go through this centralized server. To locate interested peers a query is posted to the server. After getting the result from the server, peers can communicate directly and share resources. This is the method used by Napster[10] and is not totally decentralized.

In PeerCQ, when this kind of P2P protocol is used, the centralized server is responsible for storing the CQs posted by peers and assigning these CQs to other peers for execution. Moreover the server is responsible for load balancing and detection of failures.

This approach has several advantages. Since the CQs are stored on the server, the durability of a CQ is easily guaranteed. Moreover, load balancing decisions are easy to make since the server assumes complete knowledge of the peers. Furthermore, the peer failure detection and recovery of CQ execution is easier to accomplish on the server side.

An important drawback of this kind of approach is that centralized server forms a central point of failure, and the system shares some general problems of the client/server architectures to a lesser extend.

3.2 Flooded queries

In flooded queries, a query originated from a peer is forwarded to all of its neighbors. And each neighbor again forwards this received query to its neighbors. By this way the query is propagated over the network. The decision of how far a query propagates is usually determined by a time to live field, which is decremented at each hop. During this propagation each peer that receives the query also responds to this query if it satisfies the desired properties specified in the query. This kind of P2P protocol is used in Gnutella [8].

Consider the PeerCQ system use this P2P protocol. When a peer posts a CQ to the system, this CQ is propagated to peers inside the network. Then the peers that are willing to execute this CQ respond back. Responses might also include some kind of willingness value. This value can be calculated by the responding peer based on (1) the current resource usage in the responding peer, (2) the average on-line time of the responding peer, and (3) the similarity of the CQ to the ones that the responding peer currently executes. The CQ originator peer can make its own decision of where to execute its CQ according to the returned willingness values.

A general problem with this approach is that the number of messages sent per lookup query is large and the system does not scale well.

3.3 Routed queries

In protocols employing routed queries, each peer keeps a routing table. Lookup operations are performed with the help of routing decisions based on these routing tables. In these protocols, each peer has its own protocol specific unique identifier. Similarly, each file is also assigned to a key identifier that uniquely distinguishes it from the rest of the files in the system. The lookup problem is locating the peer whose identifier is numerically closer to the key identifier in a file-lookup query. This approach is totally decentralized and has the advantage of using fewer messages for answering a query when compared to flooded queries. Protocols based on routed queries are implemented in Freenet [1], Pastry [6], Tapestry [11] and Chord [2]. They vary in their table designs, bounds on the number of messages needed for performing lookup, guarantees on successfully resolving lookups, and ability to recognize network proximity.

When this approach is used in PeerCQ, we need to assign identifiers to both CQs and peers. The protocol will determine on which peer a CQ is going to be executed by relating a CQ identifier to a peer identifier.

It is shown by previous work like Pastry [6], Chord [2], Tapestry [11], and others [1, 5, 12] that the look-up is scalable with this solution in a decentralized environment. It requires reasonable amount of space kept in peers and a small number of messages sent per look-up. In these systems, due to the general characteristics of file sharing applications, the load balance factor is defined by how well the files are distributed or how fast the lookups can be performed. However, PeerCQ system requires stronger load balancing and more complex mechanisms for fault tolerance due to complex nature of CQ domain.

All three lookup methods described above can be used in PeerCQ system by introducing different modifications. However, centralized lookup method is against our goal of building totally decentralized and self-configuring system. On the other hand, flooded query based lookup is

inefficient due to its use of large number of messages per lookup. As a result, in the first implementation of PeerCQ we extend the basic routed-query based protocol to address the needs of CQ domain.

4 Design Decisions

There are two main challenges in designing a routed query based protocol that incorporates application aware information into the service partitioning and lookup algorithms. First, the proposed protocol should preserve the decentralization and scalability in terms of lookup performance. Second, the protocol should exhibit better load balancing by taking into account both the CQ-aware information and the peer resource-availability in its service partitioning scheme.

Service partitioning in PeerCQ manages the assignment of CQs to appropriate peers, with the objective of balancing the load of the peers in the system. More concretely it addresses the problem of load balancing in terms of how to distribute CQs over peers so that the load of each peer is commensurate to the peer capacities (in terms of cpu, memory, disk, and network bandwidth). The term load balancing is usually used for algorithms that try to "equalize" the load at all nodes and is quite difficult to accomplish, especially in the presence of heterogeneous nodes in terms of server capacity and network bandwidth. Load balancing in PeerCQ is carried out by its service partitioning scheme. We use the weak concept of load balancing, which tries to reduce and avoid load peaks when distributing CQs over the peers of the system.

We propose a totally decentralized P2P architecture and a P2P protocol for distributing CQs over peers of the system. The PeerCQ P2P protocol extends the basic routed query based protocols in terms of service partitioning (CQ to peer assignments) and service lookup. The PeerCQ service partition scheme is called hashing-based randomization with peer donation. One of the salient features of our scheme is its extension of a peer-unaware randomized partition (scheduling) algorithm with both peer-aware and CQ-aware capability. Before entering a detailed discussion on this proposed scheme, we first informally describe the design ideas behind this proposed scheme.

4.1 Peer-aware Service Partitioning

Peer-awareness is an important property of a desired load balancing scheme for any P2P system that allows heterogeneity in peers, including PeerCQ. In peer-aware load balancing, scheduling decisions are based on the information coming from the peers of the system. Since PeerCQ is totally decentralized and fully distributed, there is no scheduling node in the system. A job, which is a CQ, could be posted from any peer of the system. Moreover, a peer does not have global knowledge about the rest of the peers in the system. As a result, peer-aware load balancing is an interesting and challenging problem.

Randomized scheduling schemas are easy to implement in decentralized systems. However they do not perform well in terms of load balancing in heterogonous environments. The reason is that they are not using any peer information to capture the heterogeneity found in the P2P network. A good way to achieve the desired load balancing in PeerCQ is to use a hybrid approach that combines a peer-unaware randomized scheme with a peer-aware one.

The hybrid scheme we propose is peer-aware hashing based randomization, which recognizes the diversity between peers. Concretely, we implement peer-awareness based on peer donation. Each peer donates a self-specified portion of its resources to the system. The scheduling decisions are based on the donated amount of resources.

4.2 CQ-aware Service Partitioning

Another important property of a desired balancing scheme for PeerCQ system is CQ-awareness. In CQ-aware load balancing, scheduling decisions utilize characteristics of CQs, such as similarity in trigger conditions of CQs.

Distribution of CQs to peers, which allows assignment of CQs having similar triggers to same peers, is an important consideration in a CQ-aware load balancing scheme. This is required in order to reduce the redundant processing (i.e., repeated evaluation of the same trigger condition). An interesting challenge to CQ-aware partitioning is the following situation. When the number of CQs with the same trigger condition varies significantly from trigger to trigger, some peers may end up with a lot more CQs to process than the others. This might hurt load balancing. We discuss how PeerCQ addresses this issue in the next section.

4.3 PeerCQ Service Partitioning Scheme

It is obvious that a service partitioning scheme that supports only peer-awareness or only CQ-awareness will not be able to address both peer heterogeneity and system scalability with respect to the total number of CQs. Our decision is to combine both CQ-awareness and peer-awareness into our service partitioning algorithm. By peer-awareness, each peer on the network is loaded roughly the same, and the load criteria is based on the amount of resources a peer is willing to donate to the system. By CQ-awareness, CQs are distributed to avoid duplicated processing of CQs and to achieve better system utilization. We call this scheme the hashing based randomization with peer donation.

5 Hashing Based Randomization with Peer Donation

As discussed in Section 3, PeerCQ system is based on a P2P architecture that employs a routing based P2P protocol. The assignment of CQs to peers in this protocol is based on mapping CQs and peers to identifiers in the same identifier space and matching CQs to peers based on

the numerical closeness of their identifiers. The basic routing based protocol provides no support for peer-aware or CQ-aware capabilities. We propose the hashing based randomization with peer donation scheme in PeerCQ to extend the basic routing protocol. This scheme incorporates both peer-aware and CQ-aware capability into the service partitioning algorithm, with the objective of maintaining good load balance when distributing CQs over peers in PeerCQ.

Peer-awareness is achieved by mapping peers to different number of points in the identifier space depending on the properties of peers. Concretely, each peer donates a self-specified portion of its resources to the PeerCQ system. Based on the donated amount of resources, the service partitioning algorithm maps peers with higher donations to more points in the identifier space. Note that this approach does not require global information.

CQ-awareness is achieved by mapping CQs with the same triggers to a contiguous region, instead of to the same point, in the identifier space. The responsibility of processing CQs with the same type of trigger conditions is assigned to a smaller group of peers that are mapped onto this contiguous region. This design decision is made explicitly to address the potential load peak problem when the number of CQs with the same trigger condition varies significantly from trigger to trigger, leading to the situation where some peer may end up with a much larger number of CQs than others.

There are three distinct mechanisms underlying the PeerCQ service partitioning: (1) mapping peers to identifiers, (2) mapping CQs to identifiers, and (3) matching between CQs and peers based on their identifiers. These mechanisms form an important part of the PeerCQ system that differentiates it from other P2P systems. The implementation of these mechanisms in PeerCQ is discussed in the following sections respectively.

5.1 Mapping peers to identifiers

Mapping of peers to identifiers is intended to address the peer-awareness objective by integrating peer information into the load balancing decision. The idea is to map each peer to a set of *m*-bit identifiers, which forms that peer's identifier set. *m* is a system parameter, it is large enough to ensure no two nodes are mapped to the same identifier or this probability is negligible. Given a peer, we call the set of identifiers to which it is mapped the virtual identifiers. The peers that donate more resources are assigned more virtual identifiers, so that the probability that more CQs will be matched to those peers is higher. This addresses the peer-awareness requirement of the load balancing algorithm. Figure 2 shows an example of mapping of two peers, P1 and P2, to their peer identifiers (virtual identifiers). Peer P1 has 3 virtual identifiers, whereas peer P2 has 5.

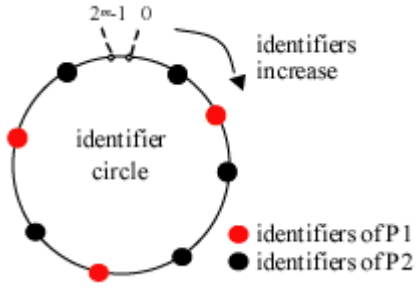

Figure 2

The number of identifiers to which a peer is mapped is calculated based on a peer donation scheme. We introduce the concept of *ED* (effective donation) for each peer in the overlay network. *ED* of a peer is a measure of its donated resources effectively perceived by the PeerCQ system. For each peer, an effective donation value called *ED* is first calculated and later used to determine the number of identifiers that peer is going to be mapped. The calculation of *ED* is given in Appendix A.

In addition to design the mapping in such a way that the probability that two peers are mapped to the same identifier is negligible, the mapping of a peer to identifiers has to be as uniform as possible. This can be achieved by using hashing functions like MD5 or SHA1 (or any well-known message digest function). The following algorithm explains how peer identifier set is formed assuming that we already calculated the effective donation, and we use *m*-bit identifier space consisting of 2^m identifiers:

```
setupPeerIDs(ED)
  peerIDs = empty
  // set of peer ids this peer possesses
  for i = 1 to donation_to_ident(ED)
   add SHA1(peerIP.port.i, m) into peerIDs
   // . is concatenation
```

The function `donation_to_ident` is responsible of mapping the effective donation value to the number of identifiers that the peer with such a donation should posses. The return value of this function is an integer in the range [1,*D*] and *D* is determined based on load balancing performance. If *D* is too large then the number of CQs assigned to a node with *D* number of identifiers might be too large, over utilizing the peer. On the other hand, if *D* is too small then the peers with high effective donations will be under utilized. *SHA1* is a message digest function. The first parameter of this digest function is the input message that will be digested, and the second parameter is the length of the output in bits. This algorithm maps each peer to a set of *m*-bit identifiers in terms of the peer IP address, port number, and the normalized donation in the range [1,*D*].

5.2 Mapping CQs to identifiers

Mapping of CQs to identifiers is intended to address the CQ-awareness objective by integrating CQ information into the load balancing decision. The goal is to map CQs with the same trigger conditions to the same peers as much as possible, in order to achieve higher overall

utilization of the system. One solution is to hash trigger identifier and CQ identifier separately and concatenate the results. This mechanism allows CQs with the same trigger to be mapped into a contiguous region on the *m*-bit identifier circle.

The length of a CQ identifier is *m*, which is a system parameter (A *m*-bit identifier space contains 2^m identifiers). The length of the first part of an *m*-bit CQ identifier is *a*, which is another system parameter called *grouping factor*. Given *m* and *a*, CQs are mapped to identifiers as follows:

```
calculateCQID(CQ)
    CQID = SHA1(trigger of CQ, a).
         SHA1(IP.port.time.rand, m-a)
    // . represents concatenation
    return CQID
```

The first digest function generates the same output for the CQs with same trigger conditions, where the second one generates a globally unique output for each CQ posted by a peer. The first parameter of the first digest function is the trigger condition of a CQ. The first parameter of the second digest function is the concatenation of IP address, port number of the peer posting this CQ, current time and a random value. The second parameters of the two message digest functions SHA1 are the lengths of the outputs in terms of bits. This mapping tries to assign CQs with the same triggers to the same peers by mapping them to a point inside a contiguous region in the identifier space. According to the parameter *a* (grouping factor) of the first function the identifier space is divided into 2^a contiguous regions. Depending on its trigger part, a CQ is mapped to an identifier in one of these regions. Since the number of CQs is expected to be larger than the number of peers, the number of CQs mapped inside one of these regions is larger than the number of peers mapped. Introducing smaller regions increases the probability that two CQs having the same trigger condition is matched to the same peer. This can be seen from another point of view, peers whose identifiers map into the same identifier region are specialized for serving a specific set of triggers. However taking into account the non-uniform nature of the trigger conditions, there is a trade-off between achieving less redundancy in trigger condition evaluation and forming hotspots. So, the parameters should be carefully chosen.

As an example consider $m = 5$, and $a = 3$. Then the total number of identifiers is $2^m = 32$. The identifier space is divided into $2^a = 8$ contiguous regions, each of which is of length 4. Assume that the first digest function hashes the trigger condition "T" into a 3-bit value 110. Then the set of possible identifiers to which a CQ with trigger condition "T" can be mapped is in the identifier interval *IdInt* = [110*00*,110*11*]. The length of *IdInt* is 4. The interval *IdInt* covers $1/2^a = 1/8$ of the whole identifier space. As a result on average 1/8 of the peers are specialized for processing CQs with trigger condition "T". If we generalize this, given that there are *N* peers, the number of peers specialized for a given trigger condition on average is $N/2^a$. Note that this does not means that peers specialized for a given trigger condition are only assigned with CQs of that type of trigger condition. There are two reasons or this. First, if the grouping factor *a* is not large enough, then two CQs with different triggers might be mapped to the same peer identifier and thus inside the same contiguous region. Second, peers might have more than one identifier possibly belonging to different contiguous regions.

5.3 Matching CQs to peers

Matching a CQ to a peer is done by mapping the CQ to its identifier and then using the underlying P2P protocol's lookup operation to find the peer responsible for processing the CQ. The job of the lookup operation is to choose the peer that will process the CQ, bearing in mind the load-balancing objective. Different criteria can be applied to define the matching. For example, using the criterion given in Consisting Hashing [7] in the context of PeerCQ, a CQ, say cq_i, is matched to a peer, say p_i, if the difference between the CQ identifier and the peer identifier, i.e., $|cmap(cq_i) - pmap(p_j)|$, is minimum over existing peers, where *cmap* is the function that maps a CQ to an identifier and *pmap* is the function that maps a peer to an identifier. Consisting hashing work [7] mathematically proves that in such a matching each peer will get $(1+\kappa)O(K/N)$ CQs with a high probability where *K* is the number of CQs, *N* is the number of peers and κ is a small constant. According to [7], to achieve such good bounds each peer has to posses $O(\log(N))$ virtual peer identifies. Chord [2] builds its P2P protocol based on Consisting hashing [7]. It shows that, the virtual peer identifier approach plays an important role in achieving good load balance. If not used the variation in the number of jobs (CQs in PeerCQ context) assigned to peers is higher.

In PeerCQ, the matching of CQs to peers is defined based on some numerical closeness of their identifiers. Concretely, a CQ is matched to a peer that has the smallest identifier in the set of peers whose peer identifiers are larger than or equal to the CQ identifier (matching criteria). This criteria can be formally described as follows: a CQ, say cq_i, is matched to a peer, say p_k, if $pmap(p_k) \varnothing cmap(cq_i)$, and for any peer p_j such that $p_j \prod p_k$ and $pmap(p_j) \varnothing cmap(cq_i)$, we have $pmap(p_k) < pmap(p_j)$.

Our extension to the virtual identifier approach is the use of different number of virtual peer identifiers for peers based on their resource donation. The virtual peer identifier method is the key point where we add the donation scheme into the picture. The peers that donate more resources are assigned more virtual identifiers, so that the probability that more CQs will be matched to those peers is higher. Furthermore, when a peer enters or leaves the system, the assignment of some CQs has to be changed according to the matching criteria. This is

important as the matching scheme plays a major role in achieving good load balance.

Several studies on variants of Consistent Hashing work [2, 6] have shown that lookup operations that help to match a key (CQ in our case) to a peer in a decentralized environment is not expensive, in the order of log(N) and the routing information that has to be stored in nodes for that purpose is also in the order of log(N). More importantly, the number of CQs that has to be moved when a new peer enters or leaves the network is $O(K/N)$.

We discuss the details of the PeerCQ P2P protocol and its lookup mechanism in the next section.

5.4 PeerCQ P2P Protocol Details

The P2P protocol we describe here supports a lookup operation, which given a key identifier returns the address of the peer responsible for that identifier. If the key identifier in the lookup query is a CQ identifier, this lookup operation returns the peer to which this CQ was assigned according to the CQ-to-peer matching criteria used in the PeerCQ service partition scheme. If the key identifier is the peer identifier, this lookup operation will locate the peer with the given peer identifier.

In this section we first describe the routing information kept at each peer for efficient routing of lookup queries. Then we describe the lookup algorithm.

5.4.1 Routing Information

Routing, which is needed for lookups, is achieved with the use of routing information maintained at each peer. In PeerCQ the routing information consists of a routing table and a neighbor list for each identifier possessed by a peer. In other words each identifier of a peer has a routing table and a neighbor list associated with it. The routing table is used to locate a peer that is more likely to answer the lookup query (routing step 1), where a neighbor list is used to locate the peer that is matched to the key identifier of the lookup query according to the matching criteria (routing step 2). Peers that have multiple identifiers will have multiple routing tables and neighbor lists, one routing table and one neighbor list per identifier.

5.4.1.1 Routing Table

Routing table is used to route the lookup queries towards their destination peers. Assuming an m-bit identifier space, the number of rows that a routing table has is $\lceil m/b \rceil$, where b is a parameter such that 2^b represents the radix of the identifier. When the m-bit identifier is represented in radix 2^b, it has $\lceil m/b \rceil$ digits. For instance if $m = 8$ and $b = 4$, an identifier is represented with two hexadecimal digits. (10110010 = B2) The routing table also has 2^b columns corresponding to integers in the range $[0..2^b\text{-}1]$. In other words, the table has one column for each possible value of a digit. The choice of b includes a tradeoff between the quality of routing and the amount of state we have to keep in a peer. Assigning larger values to b means keeping more information, and results in better routing, while it also means larger state and results in larger memory consumption and higher cost in initialization and maintenance of the routing table information.

Each entry of a table contains a pair consisting of a peer identifier and a reference, which is the IP address and port number of the peer possessing that identifier. The entries in the table obey two rules:

(1) Entries in the ith row (rows start from 0) of a table contains pairs with peer identifiers sharing i leftmost digits with the associated identifier of the peer owning the rooting table.

(2) An identifier in the jth column of the ith row of a table has j, as its i indexed digit from the left (leftmost digit is indexed as 0).

Consider an identifier space where $m = 8$ and $b = 2$. A 8-bit identifier has 4 digits ($\lceil m/b \rceil = 4$). Table 1 is an example routing table for a peer with identifier $peerID$ = 2103. (For simplicity the table shows only the identifiers)

	0	1	2	3
0	0210	1302		3002
1	2033		2213	2301
2		2113	2122	2132
3	2100	2101	2102	

Table 1

The main idea behind the routing table is to support fast routing of queries. The routing table structure makes it possible for a peer to be aware of other peers that have identifiers sharing long prefixes with its identifier. It means that, by routing a lookup query from a peer X to the destination peer Y, given that Y has an entry in the routing table of X and Y's identifier shares the longest prefix with X's identifier, the lookup query is passed to a peer, which has a better knowledge of the peers having closer identifiers to the key in the lookup query. See Section 5.4.2 for a detailed lookup operation.

5.4.1.2 Neighbor List

The routing information also includes a neighbor list for each identifier a peer possesses. Neighbor lists serve for two purposes, one is for routing and the other is related to failure handling. Neighbor lists are used to verify the matching criteria and to speed up the second step of routing faster. They also form the basis on which the failure handling mechanisms of PeerCQ is built. Each neighbor list contains r pairs, where a pair is structured in the same way as a pair in a routing table entry. A neighbor list keeps track of $r/2$ followers and $r/2$ predecessors of its associated identifier on the identifier circle. More concretely, given a peer, say p_i, its neighbor list has $r/2$ of the pairs containing peers whose identifiers are the largest ones over the set of peers whose identifiers are smaller than p_i's identifier. Similarly, its neighbor list also has $r/2$ pairs containing peers whose identifiers are the smallest ones over the set of peers whose identifiers are larger than p_i's identifier. Similar to the parameter b of routing table, the selection of r includes a tradeoff between better routing performance and cost of maintaining it.

Recall the previous routing table example, where the associated peer identifier is 2103. The neighbor list for *peerID* = 2103 can be as follows: [2101, 2102, 2103, 2113, 2120], where *r*=4 (again only the peer identifiers are shown here for simplicity).

To summarize the role of routing tables and neighbor lists in a lookup operation, consider a lookup query with a CQ identifier cq_k received by a peer *p*. If cq_k is within the neighbor list scope, the neighbor list of *p* is used to locate the peer whose peer identifier is matching cq_k according to the matching criteria (query is answered). Otherwise, the routing table is used to locate the next peer *q* that is likely able to answer the query.

5.4.2 Lookup Algorithm

Lookup is mainly achieved by forwarding a lookup query containing a key identifier to a peer whose identifier is closer to the key identifier in terms of the matching criteria discussed in Section 5.3. The result of a lookup query with the identifier *keyID* is the address of the peer who has the smallest identifier among identifiers of all existing peers that are numerically larger than or equal to *keyID*. The terms larger and smaller are in terms of numerical comparison. We use the term closer in the context of identifiers and PeerCQ matching criteria, such that id_1 is closer to id_2 than it is to id_3 means that the absolute value of numerical difference between id_1 and id_2 is smaller than the absolute value of numerical difference between id_1 and id_3.

In PeerCQ, when a lookup query reaches a peer, it usually takes three steps to take the appropriate action. Step one uses the virtual identifier list. Step two uses the neighbor list. Step three uses the routing table. Consider a lookup query with the key identifier cq_k received by a peer p_i. The first step involves selecting a virtual peer identifier, say p_{ih}, of the peer p_i, which is the closest to the key identifier cq_k assuming p_i has multiple identifiers. The second step involves checking the neighbor list of p_{ih} to see if cq_k falls inside the scope of this neighbor list. If yes, returns the address of the peer whose peer identifier is the smallest among the identifiers in the neighbor list, which are larger than or equal to the key identifier cq_k. If not, enters the third step. The third step involves selecting a peer, say p_j, from the routing table of p_{ih}, such that p_j has an identifier, which is "closer" to the key identifier cq_k and thus more likely to answer the query. Thus the query is routed to p_j. To find the appropriate table entry, the number of digits in shared prefix between the key identifier cq_k and the peer identifier p_{ih} is calculated. That number, say *l*, gives the row index in the routing table. The value of the first non-shared digit of the key identifier cq_k, say *h*, gives the column index. In case that the entry at *l*th row of *h*th column is empty the lookup algorithm finds the best peer in terms of numerical closeness by considering all entries in the routing table and the neighbor list of p_{ih}, and forward the query to that peer.

Figure 3 illustrates a full lookup process example. In the example, a lookup query with a CQ key identifier 2123 (CQ 2123 in the figure) is posted from the peer with peer identifier 0201 (P 0201 in the figure). Figure 3 shows that, it takes four steps to locate the right peer that will process this CQ. All of the first three steps are routings of the lookup query via routing tables of corresponding peers. The last step is the step in which the lookup query is answered via neighbor list lookup. In the first three steps, the lookup query is routed to a different peer using routing tables. Because, the CQ identifier in the query is not in the range of peers' (P 0201, P 2332, P 2102 respectively) neighbor lists and the peers do not satisfy the matching criteria. In the fourth step, still the peer (P 2125) does not satisfy the matching criteria, but the CQ identifier falls into its neighbor list range. As a result the peer responsible for the CQ (P 2124 in the figure) is located by the neighbor list lookup.

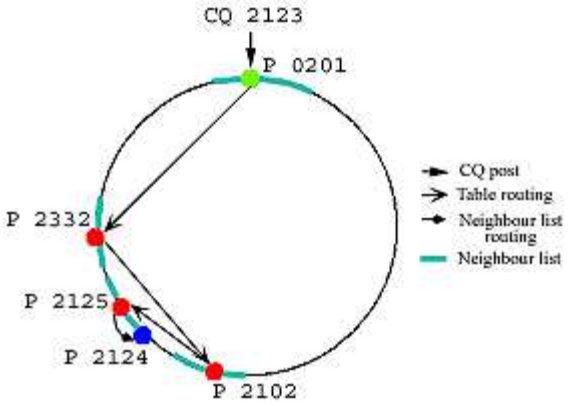

Figure 3

Due to the space limitation, we omit the complete lookup algorithm in this paper. Readers may refer to our technical report [18] for the pseudo-code of the algorithm.

5.4.3 Initializing and Maintaining Routing Information

We have discussed the structure and the usage of the routing table and neighbor list. An immediate question is how to create and maintain the routing information.

The routing information of a peer is initiated when the peer joins the system. A peer, say *P*, who wants to join the system needs to find an existing peer, say *X*, as an entry point peer. There are several ways to determine an entry peer. It can be determined by making use of well known peers that are online most of the time as done in Gnutella [8], or some other methods (like IP multicast's expanding ring search) might be used since this does not affects the routing.

We do not discuss how the routing information of the peers is initialized and maintained in this paper. It is described in our technical report [18].

5.5 Additional Properties

In addition to the peer-aware and CQ-aware capabilities, another interesting property of the PeerCQ service partitioning scheme, namely the hashing based

randomization with donation, is that a malicious peer that posts lots of CQs in order to overload a specific peer can not easily succeed in such an attempt, thanks for the nearly uniform distribution of CQ identifiers. However, posting CQs with same triggers might form an overload on some of the peers. This is due to the CQ to CQ-identifier mapping, which causes CQs with the same trigger to be assigned to a group of peers. Even in this case, it is nearly impossible to find a trigger that will be mapped to a desired identifier due to the secure hash function used for the mapping.

Another additional property of the PeerCQ service partitioning scheme is its ability to make use of the stableness information related to peers. Stableness of a peer is determined based on the average time it contributes to the PeerCQ system. (This is realized by the *rel* value in *ED* calculation as explained in Appendix A) Stableness is used for two purposes. First, it is used to decrease the number of CQs assigned to peers that are not willing to participate long enough in the system. This measurement is important because it is in general more costly to recover the CQ executions when an active peer processing them fails or departs. Second, the stableness of a peer is used to restrict the number of CQs a peer can post to the system. This is a motivation for encouraging more stable participation in the system.

6 Performance Evaluation

A major concern in PeerCQ is the effectiveness of CQ distribution (service partitioning) in terms of load balancing over peers with diverse server capacities and network connections. If the service partitioning is effective, the peers of the system are well utilized and the system as a whole can achieve higher throughputs and scale well. In order to evaluate the PeerCQ service partitioning, we have carried out several experiments in terms of different measures. This section describes our experimental setup, results, and evaluation.

6.1 Experimental Setup

For the purpose of experiment we have built a simulator that assigns CQs to peers using the algorithms described in this paper. With this simulator, we take different snapshots of the system with different numbers and sets of CQs, and also with different system parameters. Then we do our measurements on these snapshots. For the initial experiments reported in this paper, we did not model the dynamic behavior of joining, leaving and failing peers. We assume that the system is settled down at these snapshots.

For the simulation we first model the peers. A peer has resources, donation, reliability and IP address. The resource distribution is taken as normal distribution. Donation of the peers is set to default, which is donating half of their resources. Secondly, we model CQs with their trigger parts. The trigger part distribution is chosen to model the hot spots that arise in real life situations due to the popularity of some triggers.

The system parameters to be set in the simulator are:
(1) *m*, the length of identifiers in bits,
(2) *a*, the grouping factor, and
(3) *D,* the maximum number of identifiers per peer.

In all of the following experiments *m* is taken as 128 and *D* as 25. The number of peers in the system is taken as 10000. This number is determined according to measurements reported by [16] that 10000 corresponds to 50% of the total population of the peers in the Gnutella network at any time.

6.2 Performance Metrics

There are three main measures we are interested in, namely: (1) sensitiveness to peer heterogeneity, (2) effect of grouping, and (3) effectiveness of load balancing. The first one measures how PeerCQ service partition scheme reacts to peer heterogeneity. The second one measures the effect of grouping on utilization of the system. And the last one measures how well the load is balanced in PeerCQ. Note that the first two are related with peer-awareness and CQ-awareness respectively.

6.3 Sensitiveness to Peer Heterogeneity

The heterogeneity of the peers in the PeerCQ system is captured by their *ED* (effective donation) value, which is reflected in the number of peer identifiers they posses. As a result, we measure the average number of CQs distributed over peers with different number of peer identifiers. Recall that an objective of our load-balancing algorithm is to assign CQs to peers, so that the number of CQs a peer possesses for processing is proportional to the number of identifiers it has, which in turn is proportional to the resources it donates to PeerCQ.

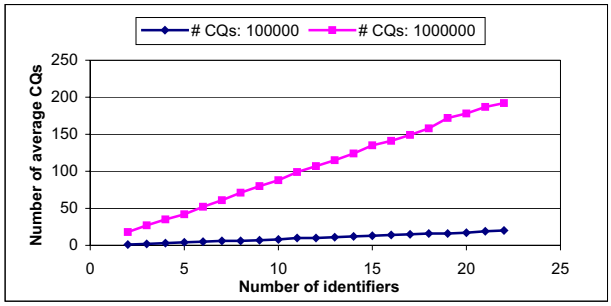

Figure 4

The values on x-axis in the graph of Figure 4, represents the number of identifiers possessed by peers, and the corresponding values in y-axis represent the average number of CQs assigned to those peers. This graph shows the matching of CQs to peers is sensitive to the amount of effective donations of peers. Peers with higher donations will be assigned more CQs.

6.4 Effect of Grouping

There are three important observations that facilitate the understanding of the results of the experiments in this

section. First, continual queries (CQs) are grouped in terms of their trigger similarities. If the number of groups is smaller, a peer can process more CQs, which means higher throughput and better scalability. Second, assigning roughly the same number of CQs to peers with similar effective donations, does not means that they are loaded roughly the same. Third, the CQ load of a peer, namely the number of CQs processed by that peer divided by the number of virtual peer identifiers it possesses, cannot be taken as the load on a peer. The main reason for that is the possible grouping in the CQ processing. Assume that peer P_1 is assigned 10 CQs and peer P_2, which has twice the number of peer identifiers P_1 has, is assigned 20 CQs. Note that their CQ loads are equal. It might be the case that the CQs in P_1 can be partitioned into 5 groups of sizes 4, 2, 2, 1, 1; where the CQs in P_2 can be partitioned into 2 groups of sizes 12, 8. In this case it is quite possible that the peer P_2 is loaded less than peer P_1, because peer P_1 has to execute 5 different triggers while P_2 needs to execute only two.

These observations have two important implications. (1) Assignment of CQs to peers that tries to achieve better grouping can decrease the average load of the peers, utilizing the overall system better. This prevents overloading on peers and leads to better throughputs when the total number of CQs increases to large numbers. (2) The variance in the number of CQs processed by peers with similar effective donations (i.e., variance in CQ loads) is not a direct indication of bad load balance in the presence of a grouping strategy.

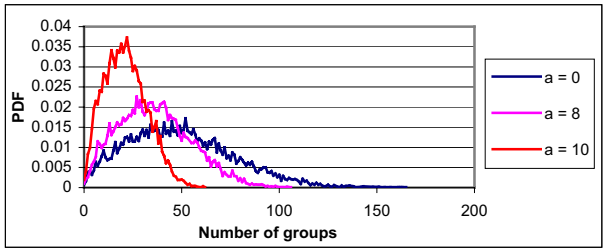

Figure 5

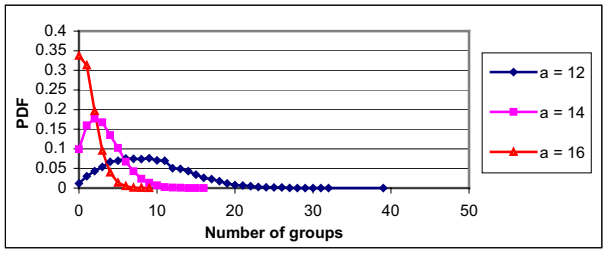

Figure 6

In PeerCQ service partitioning scheme, an important parameter that effects grouping is the grouping factor *a* (recall Section 5.2). We expect smaller number of groups per peer for larger values of *a*. However, as discussed before increasing *a* has limitations. (See Section 5.2) The graphs in the Figure 5 and 6 shows the effect of increasing *a* in terms of grouping. The values on the x-axis are the number of CQ groups that the peers have and the corresponding values on the y-axis are the probability distribution of number of peers having that number of groups. In both of the figures, the total number of CQs is 10^6 and the trigger distribution is non-uniform.

As seen from the Figure 5 and 6, higher grouping factors decrease the number of groups inside the peers and consequently utilize the peers better by taking advantage of grouping in trigger condition evaluation. However, increasing the grouping factor too much causes a lot of peers getting no CQs.

To see how the grouping affects the CQ distribution, we looked at the number of CQs distributed over peers with a given number of identifiers. The x-axis of the graphs in the Figure 7 and 8 shows the number of CQs and the corresponding values on the y-axis shows the probability distribution of number of peers which have that much CQs assigned. In both figures, the total number of CQs is 10^6. The graph in Figure 7 shows the non-uniform trigger distribution case, where the graph in Figure 8 shows the uniform trigger distribution case. In uniform trigger distribution case it is assumed that the trigger conditions are uniformly selected, where in non-uniform trigger-distribution case they are taken as to model the hot spots that occur in real life due to popularity of some triggers. The peers subject to this experiment are the ones with 10 virtual peer identifiers.

When there is no grouping, the bell shaped distribution curves in the Figure 7 and 8 shows that most of the peers get similar number of CQs. However, as the grouping factor increases this distribution is disturbed due to the non-uniformity of the CQ identifiers caused by the grouping. This is much more apparent in non-uniform trigger distribution case.

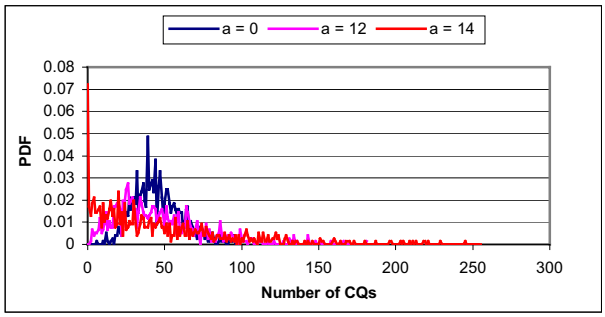

Figure 7 - non-uniform trigger distribution

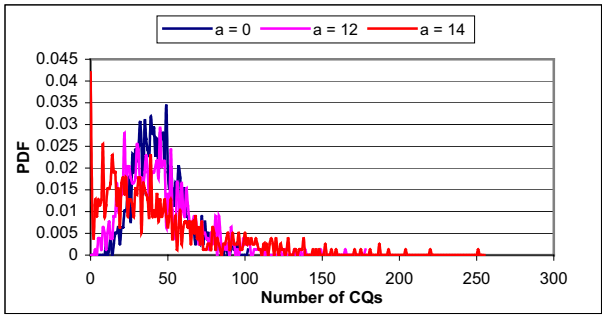

Figure 8 - uniform trigger distribution

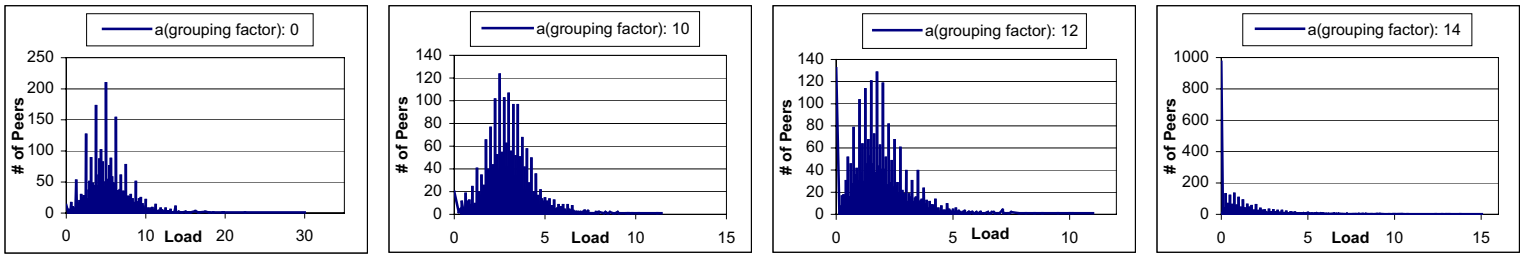

Figure 9

Although it is clear that increasing grouping factor a has implications on CQ distribution and peer utilization, the determination of a good value for the grouping factor requires a detailed measurement of load balance, which is discussed in the next section.

6.5 Effectiveness of Load Balancing

An effective measure that helps to evaluate the load balance is the load on peers. In order to analyze the load on peers we first formalize the load on a peer. Assuming most of the processing cost comes from the trigger parts, we formalize the load on a peer P as follows:

Let G_P represent the set of groups that peer P has, denoted by a vector $<g_1,\ldots, g_n>$, where n is the number of CQ groups that peer P has. We refer to n as the size of G_P denoted by size(G_P). Each element g_i represents a group in P, which can be identified by its trigger identifier. The size of a group g_i, which is the number of CQs it contains, is denoted by size(g_i). Let cost(g_i) be the cost of processing all CQs in a group g_i, tgCost(g_i) be the cost of executing group g_i's trigger, and gpCost(size(g_i)) be the cost of grouping for group g_i, which is dependent on the number of CQs in g_i. Then the cost of processing all CQs in a peer, denoted as cost(G_P), can be calculated as follows:

$$\text{cost}(G_P) = \sum_{i=1}^{size(G_P)} \text{cost}(g_i)$$

$$= \sum_{i=1}^{size(G_P)} (\text{tgCost}(g_i) + \text{gpCost}(\text{size}(g_i)))$$

For the purpose of simulation, if we assume all trigger parts are identical in terms of cost and is equal to *tgCost*, then:

$$\text{cost}(G_P) = \text{size}(G_P) * tgCost + \sum_{i=1}^{size(G_P)} \text{gpCost}(\text{size}(g_i))$$

Then the load on a peer, denoted as load(P), is calculated as follows, given we know P's effective donation ED_P:

load(P) = cost(G_P) / ED_P

In order to calculate load on peer, the cost is divided by effective donation. Because, the notion of load on a peer in our system is based on the effective donation of the peer.

Load graphs in Figure 9 are generated by taking *tgCost* = 1 and gpCost(k) = 0.25k. The values on x-axis represent the loads and the corresponding values on y-axis represent the number of peers with that much load. The total number of CQs is 10^6 and the trigger distribution is non-uniform in these experiments.

Table 2 lists several measures on the experimental data displayed by these graphs.

a (grouping factor)	mean load	load variance /mean load	CQ load variance
0	5.55401	0.77385	2.81081
10	3.02391	**0.37051**	4.38627
12	1.91490	0.50506	8.81654
14	1.39991	1.33209	24.76298

Table 2

The mean load column of the above table shows the effect of the grouping on the average load per peer. Clearly as the grouping factor increases the mean load decreases, while the variance in CQ load increases. However, as described before the increase in CQ load variance cannot be directly related to an unbalance in the load. The decrease in the mean load is desirable, but it does not reflect any information about the balance. There are two objectives of the service partitioning in PeerCQ. First, the peers should be loaded roughly the same. Second the system should be utilized well, meaning the service partitioning should be smart enough to take advantage of the grouping. As the grouping factor increases the second goal is better achieved. However if the grouping factor increases too much, the first goal will suffer. The third column of the above table shows this effect. In order to compare the balance in load for different values of the grouping factor we have to divide the load variance by the mean load, since the mean has a different value for each case. The third column shows that the balance is better when a=10.

Another interesting experiment is, how the system parameters change as the expected number of CQs and peers change. The parameter m depends on the number of peers; it should be large enough to ensure that the hash function's probability of assigning two peers to same identifier is negligible.

7 Related work

A number of systems are available for information monitoring on the web, like WebCQ [4] and Mind-It [17]. These systems are very similar to PeerCQ in terms of functionality. However, they are fundamentally different in terms of architecture. These systems are client/server

based, whereas PeerCQ employs a P2P architecture. P2P architectures are promising as a solution to general problems that arise in client/server architectures. Moreover, they proved to be successful in domains like file sharing, distributed computing, and collaborative computing. We expect that a P2P system designed for CQs will outperform a traditional client/server based approach by scaling better and requiring no central management.

Most of the current P2P applications deal with storage sharing (like [1,8,10]) or distributed computing (like [9,11]). A P2P application that we are aware of, which is related to event monitoring and notification is SCRIBE [13]. SCRIBE is a publish/subscribe based large-scale, decentralized event notification infrastructure. It uses Pastry [6] as its underlying P2P protocol and builds application level multicast trees to notify subscribers from events published in their subscribed topic. However, SCRIBE is a topic based event notification system, where PeerCQ is a generic information monitoring and event notification system. In PeerCQ notifications are generated based on the monitoring done on the web using the supplied CQs that encapsulate the interested information update requests. In SCRIBE notifications are generated from publish events of the topic subscribers.

There are several P2P protocols proposed so far, like [2,5,6,11]. Similar to work done in Pastry [6], Tapestry [11], and Chord [2], the P2P protocol described in this paper is built on top of Consistency Hashing [7] and uses some ideas described in Plaxton [12]. However our P2P protocol does not deal with caching data objects, thus it is simpler. There are two distinct features of our P2P protocol. The first is its ability to recognize the peer heterogeneity by assigning routing responsibilities, taking the diversity of the peers into consideration. The second is its incorporation of CQ-awareness into the service partition scheme to better balance the load and increase system utilization.

8 Conclusion

We have described PeerCQ, a decentralized peer-to-peer Continual Query system for Internet-scale distributed information monitoring. PeerCQ is highly scalable, self-configurable and supports efficient and robust way of processing CQs. The main contribution of this paper is the smart service partitioning with two unique characteristics. First, it introduces a donation based peer-aware mechanism for handling the peer heterogeneity. Second, it integrates CQ-aware and peer-aware information into its service partitioning scheme, while maintaining decentralization and self-configurability. We conduct a set of initial experiments, demonstrating the sensitiveness and the effectiveness of our service partitioning algorithm with respect to load balancing and system utilization.

Our work on PeerCQ continues along two dimensions. First, we are working on a number of new features to further improve the performance and reliability of the PeerCQ system, including a replication scheme for providing effective failure handling, and a trust mechanism for incorporating trust into the service partition scheme to cope with interactions with unknown or unfamiliar peers. Second, we are developing a working prototype of PeerCQ to gain more understanding on the engineering issues of building a reliable, self-configurable, and robust peer-to-peer information monitoring system.

Acknowledgement: This work is partially supported by a NSF CCR grant, and a DARPA ITO grant.

References

[1] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. *Freenet: A distributed anonymous information storage and retrieval system*. In ICSI Workshop on Design Issues in Anonymity and Unobservability, July 25-26 2000.

[2] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. *Chord: A scalable peer-to-peer lookup service for Internet applications*. In Proceedings of the ACM SIGCOMM, August 2001.

[3] L. Liu, C. Pu, W. Tang. *Continual Queries for Internet Scale Event-Driven Information Delivery*. In Special issue on Web Technologies, IEEE Transactions on Knowledge and Data Engineering, Vol.11, No.4, July/Aug 1999. pp610-628.

[4] L. Liu, C. Pu, and W. Tang. *WebCQ: Detecting and Delivering Information Changes on the Web.* In Proceedings of the CIKM 2000, Washington D.C., Nov. 7-10, 2000

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. *A scalable content-addressable network*. In Proceedings of the ACM SIGCOMM, August 2001.

[6] A. Rowstron and P. Druschel. *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*. In Middleware, 2001.

[7] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web*. In Proceedings of the 29th Annual ACM Symposium on Theory of Computing, May 1997. pp. 654--663.

[8] The Gnutella home page, http://gnutella.wego.com/

[9] Distributed.net home page, http://www.distributed.net/

[10] Napster home page, http://www.napster.com/

[11] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing*. U. C. Berkeley Technical Report UCB/CSD-01-1141, April 2001

[12] C. G. Plaxton, R. Rajaraman, and A. W. Richa. *Accessing nearby copies of replicated objects in a distributed environment.* In Proceedings of the ACM SPAA. ACM, June 1997.

[13] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. *SCRIBE: The design of a large-scale event notification infrastructure*. In Networked Group Communication, 2001. pp. 30-43.

[14] SETI@Home home page, http://setiathome.ssl.berkeley.edu/

[15] S. Saroiu, P. K. Gummadi, S. D. Gribble. *A Measurement Study of Peer-to-Peer File Sharing Systems.* Technical Report # UW-CSE-01-06-02, University of Washington, July 2001.

[16] Clip2. Gnutella Measurement Project, May 2001. http://www.clip2.com/

[17] Mind-It Homepage, http://www.netmind.com/

[18] B. Gedik, L. Liu. *PeerCQ: A Scalable and Self-configurable Peer-to-Peer Information Monitoring System.* Technical Report, Georgia Institute of Technology, Feb. 2002.

APPENDICIES

APPENDIX A

ED (effective donation) is an integer variable in the range [1,*C*]. *ED* = 1 means the effective donation of the peer is minimum and *ED* = *C* means it is maximum. *ED* represents the perceived donation of the peer by the PeerCQ system.

There are a couple of constants and variables that are used to calculate *ED*. We describe them first, and then give the algorithm for *ED* calculation:

R (resources) is a constant vector $<r_1,...,r_i>$, where $1 \le i \le 4$, representing resource types. Its value is *<"cpu", "hard disk", "memory", "network bandwidth">*. r_i is the *i*th resource type

AR (actual resources) is a vector variable $<ar_1,...,ar_i>$, where $1 \le i \le 4$, representing the amount of actual resources the peer machine possesses. ar_i is the amount of type r_i resource possessed by the peer machine. Each element ar_i is an integer, where ar_1 is the speed of the CPU in terms of MHz's, ar_2 is the amount of hard disk in terms of GB's, ar_3 is the amount of memory in terms of MB's and, ar_4 is the amount of network bandwidth in terms of Kbit/sec's.

RP (resource power) is a vector variable $<rp_1,...,rp_i>$, where $1 \le i \le 4$, representing amount of power the peer machine possesses for each resource. rp_i is the power of type r_i resource possessed by the peer machine. Each element rp_i of this vector is an integer in the range [1,5]. rp_i = 1 means that the peer is very poor in terms of resource type r_i and rp_i = 5 means it is very powerful in terms of resource type r_i. *RP* is calculated from *AR* with the use of mapping functions.

MF (mapping functions) is a constant vector $<mf_1,...,mf_i>$, where $1 \le i \le 4$, of functions where $mf_i : ar_i \Downarrow rp_i$, meaning mf_i takes as a parameter amount of actual resources of type r_i and returns the power of that type of resource. In other words $m_i(ar_i) = rp_i$.

$mf_1(x) = MIN(5, x \ div \ 400 + 1)$,

$mf_2(x) = MIN(5, x \ div \ 15 + 1)$,

$mf_3(x) = mf_4(x) = MIN(5, ilog(x \ div \ 64) + 2)$,

where *div* is integer division and $ilog(x) = \lfloor \log_2(x) \rfloor$ if $x>0$, -1 otherwise.

In practice these functions give following outputs for several real-life inputs:

AR[1] ⇓ *RP*[1]	*AR*[3] ⇓ *RP*[3]
[0, 400) ⇓ 1 , old computers	[0, 64) ⇓ 1, small mems
[400, 800) ⇓ 2	[64, 128) ⇓ 2
[800, 1200) ⇓ 3, moderate comps.	[128, 256) ⇓ 3, moderate mems
[1200, 1600) ⇓ 4	[256, 512) ⇓ 4
[1600, 2000+) ⇓ 5, pow. comps.	[512, 1024+] ⇓ 5, large mems
AR[2] ⇓ *RP*[2]	*AR*[4] ⇓ *RP*[4]
[0, 15) ⇓ 1, small disks	[0,64) ⇓ 1, dial-up modems
[15, 30) ⇓ 2	[64, 128) ⇓ 2, ISDN
[30, 45) ⇓ 3, moderate disks	[128, 256) ⇓ 3, IDSL/Cable
[45, 60) ⇓ 4	[256, 512) ⇓ 4, ADSL/Cable
[60, 75+) ⇓ 5, large disks	[512, 1024+) ⇓ 5, Cable/T1

PD (peer donation) is a vector variable $<pd_1,...,pd_i>$, where $1 \le i \le 4$, representing the donation of the peer. pd_i is the percentage of type r_i resource the peer wants to donate to the system. Each element PD_i of this vector is a real number in the range (0,1]. *PD* can be defined by the administrator of the peer and also have a preset default value.

DP (donated power) is a vector variable $<dp_1,...,dp_i>$, where $1 \le i \le 4$, representing the amount of power the peer donates. dp_i is the donated power of type r_i resource. Each element ud_i of this vector is a real number in the range (0,5] and has similar interpretation to elements of *RP* (resource power).

RI (resource importance) is a constant vector $<ri_1,...,ri_i>$, where $1 \le i \le 4$, representing the importance of each resource regarding CQ system. The elements ri_i of this vector are positive real numbers and sum up to 1.

rel (reliability) is a variable that denotes the reliability of the peer. It can be calculated at the initialization time by the equation:

rel = *MIN*(1, *average_uptime_per_session* / *expected_uptime*),

where *average_uptime_per_session* is the average time the peer participates each time it joins to the system. It can be updated on each exit or incrementally while running. *rel* is set to 1 if it is the first time for this peer to join the system. *expected_uptime* is a time considered as 'reasonable' to participate in PeerCQ system.

Then the *ED* (effective donation) is calculated as follows, given a peer *P* and its *PD* (peer donation) and *AR* (actual resources):

```
calculateED(P, PD, AR)
  ED = 0
  // i stands for the four types of resources;
  // cpu, memory, hard disk, network conn.
  for i = 1 to 4
        RP[i] = MF[i](AR[i])
        DP[i] = PD[i]*RP[i]
        ED = ED + RI[i]*DP[i]
  ED = ⌈ P.rel * (C/5) * ED ⌉
  return ED
```